AIAA-98-4533

USING ABSTRATION TO ISOLATE HARDWARE IN AN OBJECT-ORIENTED SIMULATION

P. Sean Kenney, Richard A. Leslie, David W. Geyer,
Michael M. Madden*, Patricia C. Glaab, Kevin Cunningham*

Unisys Corporation
NASA Langley Research Center
Mail Stop 169
Hampton, VA 23681

Abstract

A common problem faced in the design of an object-oriented simulation is how a complex simulation model should interface with the simulator hardware. This paper describes a design that isolates the hardware interface from the complex models of a simulation environment. A detailed description of the design is provided and the advantages and disadvantages of the design are discussed. A working example of the abstraction as implemented in the Langley Standard Real-Time Simulation in C++ (LaSRS++) framework is also presented. Conclusions drawn from the experience of implementing the design are also given.

Introduction

Interfacing a complex simulation model with the associated simulator hardware is a common problem faced in the design of an object-oriented simulation. The resulting design should result in an interface that decouples the model from the simulator hardware in the system. A tightly coupled design unnecessarily complicates a system because a class becomes harder to comprehend, modify, or debug by itself.[1]

A desirable object-oriented design couples classes with commonality through inheritance while decoupling unrelated classes. Inheritance is one method to promote code reuse through common interfaces. A good design is also complete. This means that the interface of the class encapsulates all of the meaningful behaviors of the class. Finally, a desirable design creates classes that are unit testable. This allows the users to easily verify that a class is functioning properly and to easily test modifications.

The hardware abstraction presented in this paper is the product of an iterative object-oriented design process. The design provides a decoupled, unit-testable, and complete interface to a simulation framework. The abstraction is intended to simplify the complexity of connecting simulation models to simulator hardware devices.

The Hardware Abstraction

The hardware abstraction is composed of three main components: drivers, interfaces, and builders. The drivers are the classes that actually transmit and receive data with the simulator hardware devices, the interfaces are communication classes that pass data between a simulation model and a driver, and the builders construct all of the appropriate drivers and interfaces as needed. Figure 1 uses the Unified Modeling Language (UML) to demonstrate the relationships between the drivers, the interfaces, and the simulation models.

---

* Senior Member, AIAA

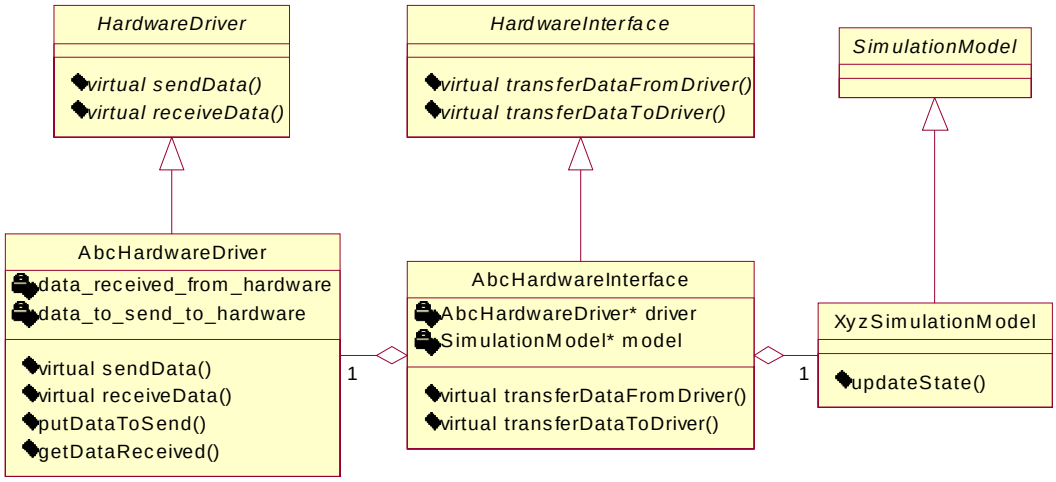

Figure 1 - Drivers, Interfaces, and Simulation Models

Drivers

The driver classes are the classes that actually transmit and receive data with the simulator hardware devices. They typically contain buffers to hold the data that is transferred with the hardware and member functions to access or modify the data buffers. In the above diagram, the class AbcHardwareDriver is shown to have defined the two virtual functions found in the abstract class HardwareDriver that send and receive data. The class also defines methods that access and modify data transferred to and from the hardware. The driver class therefore provides the abstract interface of HardwareDriver and an interface specific to the "Abc" hardware. The driver classes are an implementation of the Bridge design pattern.

The Bridge pattern decouples an abstraction from its implementation.[2] This pattern is used whenever the implementation is to remain hidden from a client[†] and the particular implementation is selected at run-time. The pattern also allows the abstractions and the implementations to be extended through subclassing.

Interfaces

Interfaces are communication classes that pass data between the driver and the simulation models. The interface class also performs any manipulation of the data before transferring the data to its destination. The interface class is essentially a one way or two way data pump between the driver and the simulation model. In the illustration above, the AbcHardwareInterface is given a reference[‡] to the AbcHardwareDriver and the XyzSimulationModel when it is instantiated. The class would then use the two references to transfer data between the two classes when appropriate. The interfaces are a variation of the Mediator design pattern.

The Mediator design pattern keeps classes from referring to each other explicitly and encapsulates how the set of classes interact.[2] The strongest asset of the Mediator design pattern is that it completely decouples the two classes from each other. It should be used whenever two classes are unrelated but need to communicate with each other.

Builders

Builder classes construct all of the appropriate drivers as requested by the user and construct all of the corresponding interfaces required by the simulation models. The builder classes provide an interface for creating the driver and interface classes without other

[†] Any object or function that operates on an object is a *client* of the object.

[‡] Unless stated otherwise, reference refers to both the reference and pointer types in C++.

simulation classes having knowledge of the particular concrete classes. The builder classes could be implemented as an Abstract Factory design pattern, a Factory Method design pattern, or any other creational design pattern.

The Abstract Factory design pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. The Factory Method design pattern defines an interface for creating an object where the subclasses decide which class to instantiate.[2] Many other creational design patterns exist. The most appropriate pattern should be used for a particular application.

Design Advantages

The advantages of this design are:

1. *The simulation models are completely decoupled from the hardware driver classes.* This allows the models to be tested with or without simulator hardware. The behavior of the simulation models due to different inputs can be fully tested offline allowing comprehensive analysis of performance without using valuable simulator hardware resources. Once the performance of a model has been validated, the hardware inputs/outputs can be used to validate the model's performance in the simulation. This minimizes the validation required of new models. The decoupled simulation models also remain portable. A class hierarchy with an abstract interface defined by the base class allows different computation models to be incorporated into the simulation and use the existing hardware interface class without modification. The model classes may be exported to other sites without requiring any modifications for use. A model imported from another site can be "wrapped" in a class that has the interface required by the existing hardware interface, thereby quickly assimilating the new computational model into the simulation.

2. *Modifications to simulator hardware only require a change to the driver class*. Many hardware devices receive major and minor modifications over their lifetimes. Minor modifications are often changes to software, buffer sizes, etc. and require little or no change to the software used to communicate with the device. Major modifications may require a significant change to the software used to communicate with the device however. Because the driver encapsulates all of the code involved with communicating to the device, the simulation is completely isolated from the modifications.

3. *The hardware driver classes can be unit tested.* Any modifications to a driver class can be tested without the simulation model. A diagnostic program can be written that uses the driver to communicate with the hardware. The diagnostic program serves two functions. It can be used to test any changes made to the driver program and it can be used to verify the operation of the hardware prior to use by the simulation. Software configuration management eases the burden of testing the new driver class by allowing the user to verify that the hardware is operating correctly with a previous version of the driver before testing the new version. (This is usually not possible when the hardware has been modified).

4. *The driver and/or interface class may be used to emulate the hardware*. Often a simulation uses real-world hardware like a flight management computer to assist in research or testing. A software emulation of a hardware device can be placed in either the driver or interface class to allow the simulation to perform necessary communications with the emulated hardware when the real hardware is unavailable. The hardware emulation may also be modified to conduct research experiments.

5. *Changes to the models can not affect the communication between a driver and it's respective hardware*. A modification to a simulation model may result in bad data being transmitted to a hardware device, but it can not cause a connection loss or crash if the hardware interface class properly limits the data being sent to the hardware device.

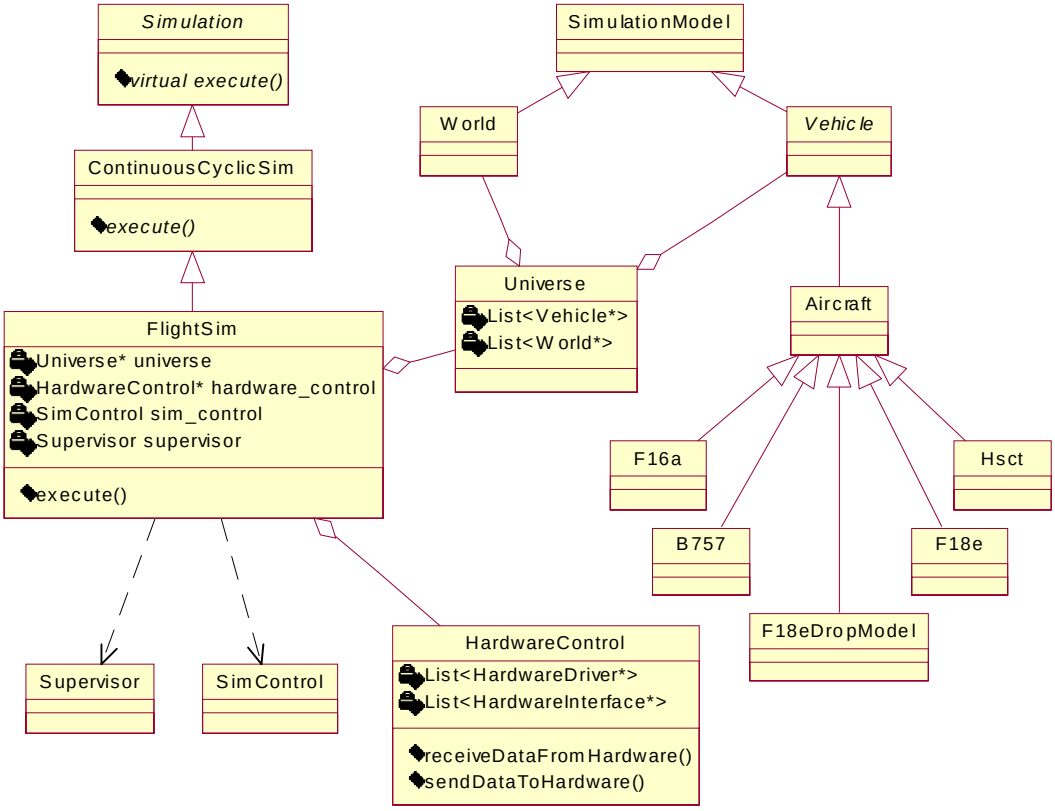

Figure 2 - LaSRS++ Framework

6. *The hardware interface classes are generally very trivial*. The classes simply use the accessor and modifier methods of the driver class and the simulation model to transmit data. Any calculations required when manipulating the data can easily be verified through testing.

7. *The hardware interface classes can often be reused by different simulation models without modification*. If the models share a common base class and the hardware interface only uses a reference to the base simulation model then no modification is required for use with different simulation models.

Design Disadvantages

The disadvantages of this design are:

1. *A modification to the interface of a simulation model requires the hardware interface class to accommodate the change*. Changing the interface of a class will always require the modification of any other classes that uses the aforementioned class. This is expected to present problems and cannot be avoided in any object-oriented design.

2. *A change to a hardware driver interface will require the appropriate hardware interface classes to be modified*. Again, this is the type of problem that is hard to avoid in a good object-oriented design.

3. *Public methods are required in model classes for all data needed by a hardware interface*. This requirement may force the designer of a class to add additional member functions to the class solely because a hardware interface needs the data. Usually a data item that is an output of a model already has public member functions to allow the class to be unit tested, so this require-

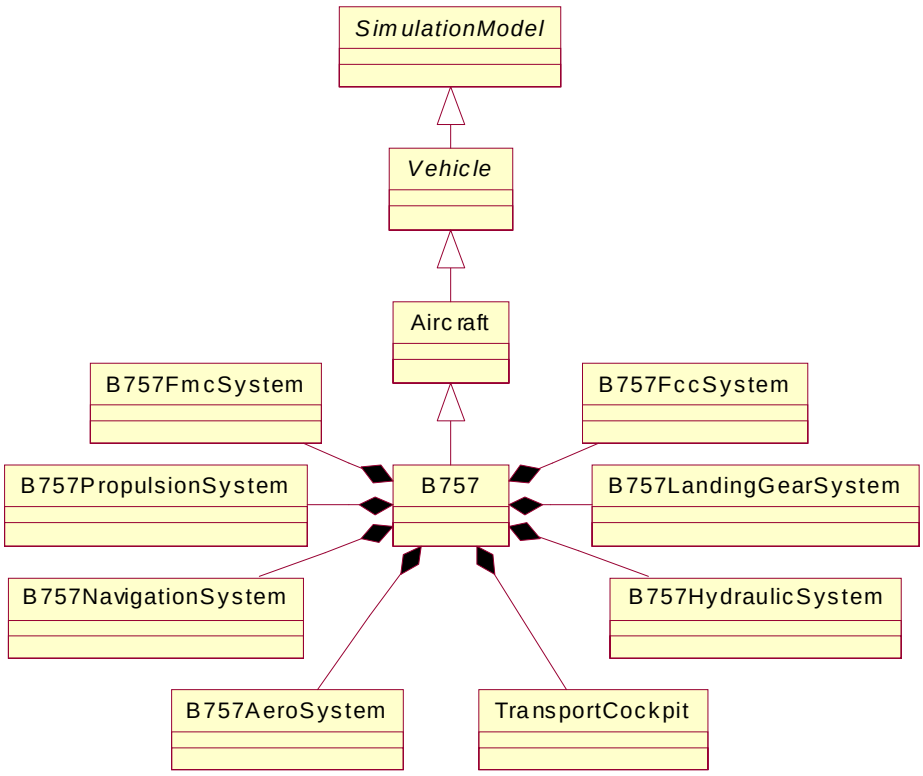

Figure 3 - Typical LaSRS++ Aircraft Hierarchy

ment is rarely a problem in a good object-oriented design.

4. *There is an added overhead associated with the interface class.* While the design decouples the driver classes from simulation models, it burdens the simulation with additional compute time to transfer the data between the two systems. Compilers provide the ability to minimize this overhead by inlining member functions.

The LaSRS++ Framework

The hardware abstraction presented in this paper is implemented in the Langley Standard Real-Time Simulation in C++ (LaSRS++) framework. LaSRS++ provides a powerful object-oriented framework for dynamic vehicle simulations in real-time, flight simulations in particular. The framework's object-oriented design makes the software extremely flexible, easily maintainable, and provides a high degree of reuse.[3] The framework is also portable and allows the user to run in either a hard or soft real-time environment.

LaSRS++ was designed to provide an efficient means to simulate dynamic vehicles of any level of fidelity. The framework allows n vehicles to be simulated on m CPUs and each vehicle may or may not be connected to a simulator hardware device.

Figure 2 illustrates several of the key components of the LaSRS++ framework. The framework supports real-time flight simulation through a class called FlightSim. This class manages the main event loop and instructs several objects to perform certain operations at specific times. FlightSim has references to HardwareControl and Universe and uses the Singleton classes SimControl and Supervisor. The Singleton creational pattern ensures that only once instance of class exists and provides a global point of access to this instance.[2] SimControl contains information about the simulation such as the current mode, the time

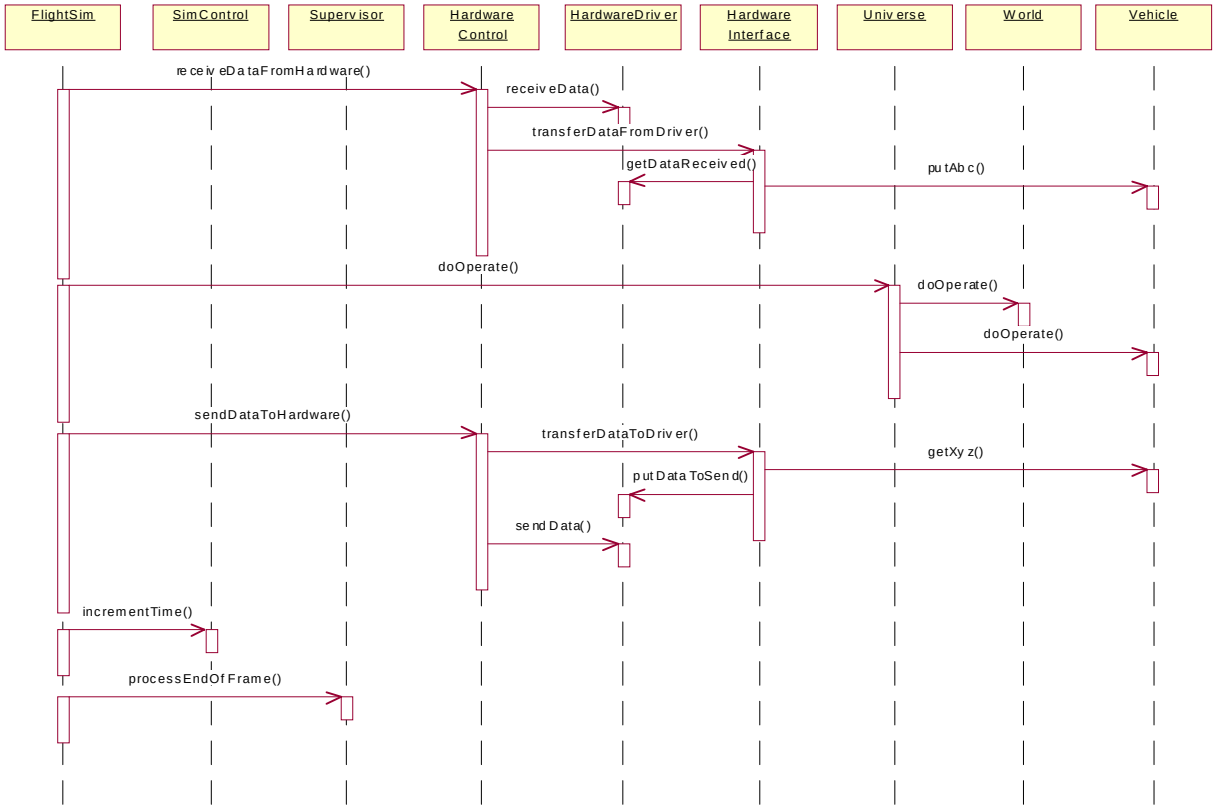

Figure 4 - Object Interaction Diagram for FlightSim

step, the current time, etc… Supervisor enforces that the simulation adheres to hard real-time when applicable. HardwareControl contains lists of all of the HardwareDrivers and HardwareInterfaces created for a particular simulation. Universe contains a list of Worlds and a list of Vehicles.

Figure 3 illustrates a typical aircraft hierarchy. The aircraft is composed of the different systems that make up the aircraft model. The illustration shows the B757 having systems for aerodynamics, propulsion, navigation, hydraulics, the landing gear, the flight management computer, the flight control computer and the cockpit. Any one of these systems may be used by a hardware interface class to obtain data from or provide data for a simulator hardware device. A hardware interface class may be passed references to the necessary aircraft systems when it is instantiated.

The simulation works in the following manner. First the drivers are constructed from scheduled hardware information by the hardware driver builder. The drivers are then placed on the hardware driver list. Next, the user selects what kind of vehicle to create, what the initial conditions of the vehicle are, which hardware the vehicle should establish a connection with, where the vehicle is located, etc… The new vehicle is then created and placed on the vehicle list. The vehicle's hardware interface builder now creates the appropriate hardware interfaces as selected by the user for the new vehicle and places them in the hardware interface list. Each hardware interface is passed a reference to the required simulation model and the appropriate hardware driver as it is instantiated. Additional vehicles and their hardware interfaces may also be created by the user in the same manner.

Figure 4 is an object interaction diagram for FlightSim and the key components of the simulation that it

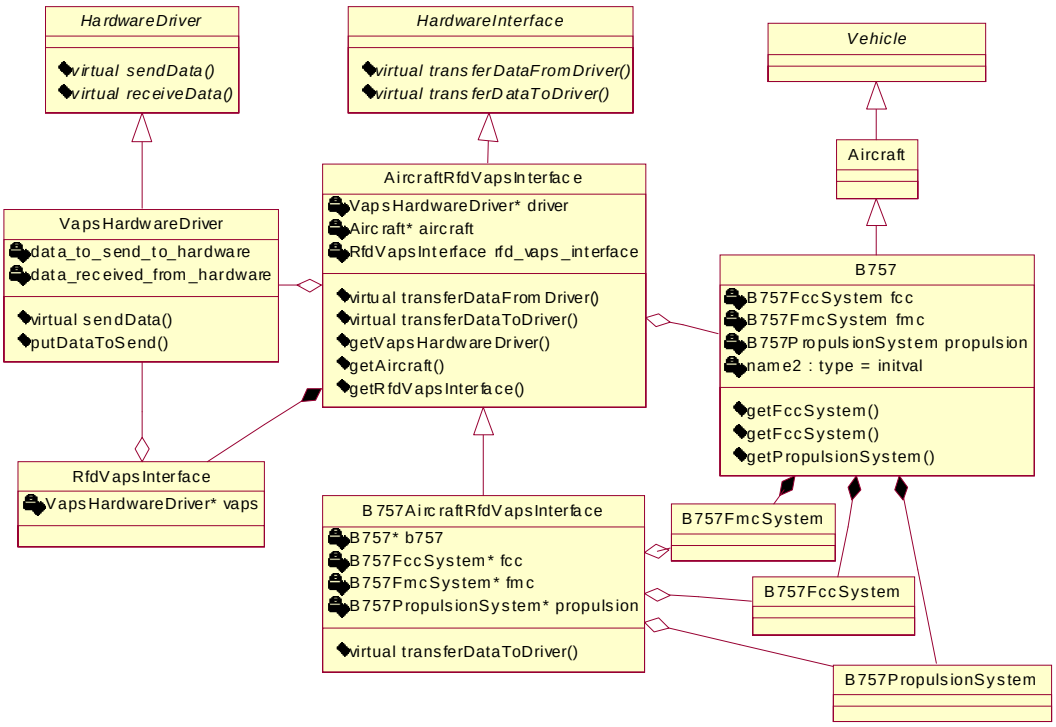

Figure 5 – A Typical LaSRS++ Hardware Interface

communicates with. At the beginning of any frame the first task of the simulation application is to march through the list of drivers and instruct the drivers to receive new data from the hardware. Next, the simulation steps through the interface list and instructs each interface to transfer data from the driver to the simulation models. The vehicles are then told to perform whatever computations they need to perform. Next, the hardware interfaces are told to access the new simulation model data and transfer it to the hardware drivers. Finally, the drivers send the data to the hardware devices. The process repeats until the simulation session is terminated.

The LaSRS++ Implementation

Recall that figure 3 illustrates the HardwareInterface classes exchanging data only with the Vehicle class. It should be noted that the hardware interface classes are capable of transferring data with not only the Vehicle classes but also with derived classes like Aircraft and B757 and any objects that these classes might contain if public accessor methods are made available. Figure 4 illustrates several of the component classes that make the B757 aircraft model. A hardware interface class specific to B757 can contain references to any of the components of B757 that it has access to. This allows the interface to directly access or modify the component system directly.

A typical LaSRS++ hardware interface is shown in Figure 5. The diagram illustrates the components used to communicate with the heads-down displays in the Research Flight Deck (RFD). The RFD is an aircraft cockpit at LaRC designed to facilitate B757 research projects. The heads-down displays used in the RFD were written with a program called VAPS (Visual APplicationS builder). The VAPS programs receive data to update the displays through a shared memory segment that is broken into channels. The VapsHardwareDriver class is responsible for allocating the shared memory segment needed to provide the VAPS program with new data.

The RfdVapsInterface class provides the means to separate the shared memory segment into the chan-

nels that the RFD heads-down displays require and the class also provides modifier functions to update the data in the channels. AircraftRfdVapsInterface class actually contains an RfdVapsInterface object along with a reference to the Aircraft for which it was created. The transferDataToDriver method uses the Aircraft reference to obtain all of the data required by RfdVapsInterface that is common to the Aircraft class. Angle of attack, mach, roll angle, pitch angle, and climb rate are examples of the data obtained from the Aircraft reference. The AircraftRfdVapsInterface also provides methods to allow the RfdVapsInterface object to be manipulated by a child class like B757RfdVapsInterface.

B757RfdVapsInterface is the B757 specific interface for the RfdVapsInterface. The transferDataToDriver method first calls the same method found in it's base class and then obtains data from B757 component systems to update any parameters in RfdVapsInterface that were not updated in AircraftRfdVapsInterface.

The design allows any aircraft to be used in a simulation in the RFD cockpit with the default heads-down displays without any modification. Because AircraftRfdVapsInterface uses a reference to Aircraft to obtain data to update the displays, the class may be used by any dynamic vehicle that inherits from Aircraft. This feature increases code reuse in the simulation framework.

The design also allows a child class to override the default behavior found in the parent class. B757VapsInterface can modify any of the data sent to the heads-down displays by AircraftRfdVapsInterface simply by resetting the data member to the desired value. This allows different units or values computed differently to be sent to the same display.

The hardware abstraction allows testing without the hardware in either hard or soft real-time (batch). Batch provides a means to test and debug modifications without tying up hardware resources. As mentioned above, the appropriate drivers are constructed from scheduling information. This allows the user to select which hardware interfaces are constructed at run time. In batch, hardware interfaces and drivers are exercised but in most cases no data is actually transferred with a hardware device because the drivers do not attempt to communicate to the hardware devices. Some of the drivers are configured to communicate with a hardware device using internet sockets to allow testing in batch. This reduces some of the hardware resource requirements while allow the simulation to communicate with a selected hardware device. In hard real-time, data transfers can also be turned off so that the hardware interfaces and drivers are exercised but no data is actually transferred with the hardware.

Cockpits

The cockpit hardware interfaces are treated slightly differently in the LaSRS++ framework. The interface class for many simulator hardware devices, out the window visuals for example, are not appropriate for containment by a vehicle in an object-oriented design. On the other hand, an aircraft certainly "has a" cockpit. To address this problem, the LaSRS++ framework provides each aircraft with a cockpit interface and allows any aircraft to run from any cockpit without the vehicle being aware of which cockpit it is connected to. An aircraft is given a cockpit interface when it is created. Currently an aircraft can have a TransportCockpit, a FighterCockpit, or a DropModelCockpit. A specific hardware interface for each cockpit interface determines how a specific cockpit behaves for a cockpit interface.

This design allows any aircraft to connect to any cockpit once the specific hardware interface for the aircraft's cockpit interface has been created. Duplication of code is avoided by placing common code in generic classes and having the specific hardware interface contain the generic class internal to themselves.

Conclusions

Because the hardware interface is abstracted away in a manner that keeps the actual simulator hardware communications hidden from the rest of the framework, the framework is a robust, portable, and easy to maintain simulation system. Continual reuse of the hardware abstraction ensures that new hardware in-

terfaces can be easily added into the framework and that these interfaces can be easily tested and debugged with or without the hardware. The advantages of the design far outweigh the few disadvantages presented here. Although the abstraction was originally designed to support the NASA Langley Research Center flight simulation hardware, the design could be used in any object-oriented framework to heighten reuse and maintainability.

Bibliography

[1] Grady Booch. *Object-Oriented Analysis and Design*. Benjamin/Cummings, Redwood City, California,1994.

[2] Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[2] Bruce Eckel. *Thinking in C++*. Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

[3] Richard A. Leslie, et al. *LaSRS++ An Object-Oriented Framework for Real-Time Simulation of Aircraft*. Paper Number AIAA-98-4529, August, 1998.

[4] Michael Madden, et al. *Constructing a Multiple-Vehicle, Multiple-CPU Using Object-Oriented C++*. Paper Number AIAA-98-4530, August, 1998.

[5] Patricia Glaab, et al. *A Method to Interface Auto-Generated Code into an Object-Oriented Simulation*, Paper Number AIAA-98-4531, August, 1998.

[6] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, third edition, 1997.

[7] John Lakos. *Large-Scale C++ Software Design*. Addison-Wesley, Reading, Massachusetts, 1996.

[8] Robert C. Martin. *Designing Object-Oriented C++ Applications Using The Booch Method*. Prentice-Hall, Englewood Cliffs, 1995.

[9] Scott Meyers. *Effective C++*. Addison-Wesley, Reading, Massachusetts, 1992.

[11] Scott Meyers. *More Effective C++*. Addison-Wesley, Reading, Massachusetts, 1996.

[12] David R. Musser, Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, Reading, Massachusetts, 1996.

[13] Terry Quatrani, *Visual Modeling With Rational Rose and UML*, Addison Wesley, Reading, Massachusetts, 1998.